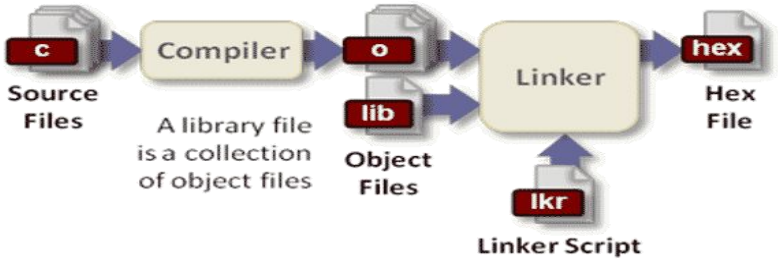
代码编译相关问题小结

ICMedia 卢立志 ——2015 年 9 月 9 日

编译过程及原理



编译过程包括：预处理、编译、汇编及链接四个部分。

1）预处理：预处理相当于根据预处理指令将源代码文件组装成新的源码程序（GCC 中-E 选项将预处理的结果写入 stdout）。预处理指令以#开头，包含：文件包含（#include）、条件编译（#ifdef、#ifndef、#endif 等）、其他（#error、#progma）等；

2）编译：将预处理得到的文件翻译成汇编代码（.s 文件）；

3）汇编：将汇编文件翻译成机器指令，并打包成可重定位目标程序的目标文件（object 文件，后缀为.o），该文件是二进制文件，字节编码是机器指令；

4）链接：将一个个的目标文件（或许还会有若干程序库）链接在一起生成完整的可执行文件。

下面以一个简单的 Hello World 来说明这个过程。程序的代码如下：

```
/**./include/hello.h**/
/**文件位于 include 子文件夹下**/
#ifndef HELLO_H
#define HELLO_H
#ifdef __cplusplus
extern “C”{
#endif
void helloFunc(char * name);
extern int g_val;
#ifdef __cplusplus
}
#endif
#endif /* HELLO_H */

/**./src/hello.c**/
/**文件位于 src 子文件夹下**/
#include <stdio.h>
#include "hello.h"
int g_val=0;
void helloFunc(char * name)
{
   printf("Hello %s!\n",name);
}

/**main.cpp**/
#include <iostream>
#include "hello.h"

using namespace std;

int main(int argc, char** argv)
{
   char name[20]="VPU";
   helloFunc(name);
   return 0;
}
```

其对应的编译过程如下：

1）采用下面的命令可以将 hello.c 编译生成 hello.o 目标文件（跳过预处理及编译过程）：

gcc -I./include/ -c ./src/hello.c -o hello.o

-I./include/表示将./include/路径包含到头文件的搜索路径中，-c 表示编译成目标文件，-o 规定输出的文件名。

由于源文件 hello.c 为 C 语言代码，需要用 gcc 按照 C 语言的规则进行编译。hello.o 目标文件中包含了 helloFunc 函数的定义，其中有类似_helloFunc_这样的标识信息用来标记函数的入口（可重定位）。

这里需要特别说明一下：在 C++编译环境下，这里的标识信息类似于_helloFunc_char*_这种形式。也就是说对于 C 语言，生成的目标文件中函数的标识由函数名表示；对于 C++来说，一个函数的完整函数名包含函数名及形参类型。详细信息可以了解 C++函数重载部分内容。

2）采用下面的命令将 main.cpp 编译生成 main.o 目标文件：

g++ -I./include/ -c main.cpp -o main.o

当编译器编译到 helloFunc 函数调用处编译器会查看能否在本文件中找到函数的定义式或者声明式。由于在 main.cpp 前面包含了头文件 hello.h，里面有 helloFunc 函数的声明，这里在目标代码中插入标识信息说明需要调用 helloFunc 函数。如果找不到 helloFunc 函数的声明或定义则会报函数未定义错误。

3）采用下面命令将 hello.o main.o 链接成可执行文件：

g++ main.o hello.o -o hello

根据_helloFunc_(由于 hello.h 头文件中 extern“C”的关系，这里的标识为 C 语言格式)标识将函数调用及函数声明处的目标代码链接到一起，形成可执行文件。

头文件的作用

头文件在整个编译过程中起到非常重要的作用，其作用具体有如下两个方面：（1）作为库文件对外的接口。在很多场合，源代码不便（或不准）向用户公布，只需向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库函数功能，而不必关心接口怎么实现。对于只在源代码内部使用的函数不要在头文件中声明，以免暴露接口。只需要在源代码文件中定义函数以供内部使用。基于这个理由考虑为了更好地服务于用户以及更好地隐藏实现的细节，需要仔细及合理地设计程序的头文件。（2）头文件能加强类型安全检查。如果某个接口被实现或者被使用时，其方式与头文件中的声明不一致，编译器就会指出错误，这一简单的规则能大大减轻程序员调试，改错的负担。定义函数的源代码文件中应该包含该函数的头文件，将提供函数声明的头文件包含在定义该函数的源文件中，以使编译器能够检查该函数的定义及声明是否一致。

头文件编写应该注意的问题：

1）防止重复包含

所有的头文件都需要采用#ifndef ... #define ...来防止被重复包含；

2）采用 extern “C”允许 C/C++混合编程

从前面的介绍中可知 C/C++在编译过程中生成的目标文件的函数定位的标识是不一样的。为了在 C++环境中正确地调用 C 语言编写的代码，这里需要用 extern “C”来确保程序正确链接。在 C++的编译器中会默认存在__cplusplus 宏定义，可以用来区分 C 及 C++的编译器。在头文件中用如下方式可以保证头文件在 C/C++条件下都能正确使用。

```
#ifdef __cplusplus
extern “C”{
#endif
    /*header info*/
#ifdef __cplusplus
}
#endif
```

这里需要用宏__cplusplus 进行条件编译是因为在 C 语言的编译器中不能正确地解析 extern “C”语法，从而导致错误。

3）包含头文件

用#include<filename.h>格式来引用标准库的头文件(编译器将从标准库目录开始搜索)，用#include”filename.h”格式来引用非标准库的头文件(编译器将从用户的工作目录开始搜索)。

4）不提倡使用全局变量

尽量不要在头文件中出现 extern int value 这类声明。

5）头文件中只存放“声明”而不存放“定义”

除了必要的内联函数以外不应该在头文件中存放函数及变量的定义，头文件在多处包含会造成重复定义错误。（“源文件的作用”部分解释）

6）内联函数

对于代码短小且频繁调用的函数可以采用内联的方式去掉函数调用过程，以避免函数调用造成的开销。内联函数只是对于编译器的一个建议，编译器可以根据自己的判断来决定是否采用内联。由于内联函数的展开式在产生目标代码的阶段，而不是在链接阶段，内联函数的定义对于编译器必须是可见的，以便编译器能够在调用点内联展开该函数的代码。因此，

不同于其他函数，内联函数在头文件中定义，以 inline 关键字表示。

inline void helloFunc（）{printf(“Hello Ingenic!\n”);}

7）头文件中不要出现 using 命令

如果头文件（C++情况下）中需要引用其他命名空间中变量、函数等不要采用 using 声明的形式，而是采用命名空间：：变量名的形式。using 声明可以方便在头文件中的编程，但在头文件中采用 using 声明，相当于在每个 include 这个头文件的文件中都采用了 using 声明，不是我们期望的结果。例如，需要引用 x265 命名空间中的 mv 变量，可以采用 x265：：mv 的形式。（附录 B 给出了命名空间的简单介绍）

源文件的作用

源代码文件中保存了程序的实现（函数及变量的定义），称为定义文件。每个源代码文件在编译的时候都会被编译成对应的目标文件。链接器根据前面所述的标识将各个目标代码链接起来形成可执行文件。从这个过程可知，各个目标代码的标识必须是唯一的，也即全局的函数及变量名称必须唯一，反过来说一个标识必须对应唯一一份实现。

将程序的实现放在头文件中，在某些情况下可以编译通过。但是当此头文件被多次包含的时候就会出现变量及函数重复定义的错误。（此时一个标识对应了编译出来的多个目标程序）因此，编写程序的时候将程序的实现代码放在头文件中实现不是一个很好的习惯！

个人的一个小建议：尽量不要在程序中 include 含有程序实现的代码文件，实在避免不了，或者包含有程序实现的代码文件会大大简化编程的情况下可以考虑将对应的文件保存成.hpp 后缀的文件。hpp 是 C++ builder 所支持的一种后缀，在公司的 Emacs 开发环境中打开 hpp 后缀的文件一样存在 C/C++关键字高亮显示等特性。通过保存成.hpp 格式以和正常的文件进行区分。

任何情况下都不要将含有程序实现代码的文件保存成.h 格式，更不要在.h 头文件中包含此种含有程序实现代码的文件！！！

Makefile 简介

上面的例子中展示了用简单的命令行指令对源代码进行编译的过程。这适用于简单的工程，对于复杂的工程需要采用一些工程管理的工具来维护整个工程的编译规则、编译顺序。下面对 Makefile 的规则进行简单介绍，一个 Makefile 大概有下面三个部分：

```
Target：Prerequisites
    Command
...
```

Target 称为目标，可以是一个可执行文件，也可以仅仅只是一个标签（label）。
Prerequisites 称为生成目标所需的依赖，可以是文件，也可以是目标（target）。
Command 是 make 程序需要执行的命令（任意的 shell 命令）。
伪目标：伪目标不是一个具体的文件，只是一个标签（label），采用 make label 命令可以使得程序执行对应的命令。

1）一个最简单的 Makefile 文件

用下面的命令可以一次完成对上面 hello 工程的编译(g++可以正确编译 C 程序)：

```
g++ -I./include/ main.cpp ./src/hello.c -o hello
```

将此编译命令写到文件中形成最简单的 Makefile 文件如下：

```
##Makefile1##
hello:main.cpp ./src/hello.c ./include/hello.h
    g++ -I./include/ main.cpp ./src/hello.c -o hello

clean:
    @echo “clean...”
    rm -rf main.o hello.o hello
```

上面的 Makefile 文件中一共有两个目标：hello、clean。冒号左边的 hello 为目标，冒号右边 main.cpp ./src/hello.c ./include/hello.h 三个文件为生成目标的依赖。hello 为文件中的第一个目标，是这个 Makefile 的默认目标（第一个目标自动成为默认目标）。在当前路径下执行 make（make hello）可使程序运行下面的命令 g++ -I./include/ main.cpp ./src/hello.c -o hello。

clean 是第二个目标，是一个伪目标，之所以称为伪目标是因为不存在一个名字为 clean 的文件。在命令行中执行 make clean，可以运行下面的两条命令。（每条命令都需要以【tab】键开头，@表示不在 shell 中显示当前命令）

进一步解释依赖：hello 的生成依赖于 main.cpp ./src/hello.c ./include/hello.h 三个文件。当执行 make（make hello）的时候，make 程序会查看当前目录下是否存在 hello 文件，其时间戳是多少，查看三个依赖文件的时间戳。当 hello 文件不存在，或者 hello 文件的时间戳早于 main.cpp ./src/hello.c ./include/hello.h 三个文件中的任何一个文件的时间戳，则执行命令 g++ -I./include/ main.cpp ./src/hello.c -o hello 生成新的 hello 可执行文件。这时候再一次执行 make 的时候，由于 hello 的时间戳晚于其所依赖的三个文件的时间戳，则不需要执行下面的生成命令。make 程序通过这种依赖机制来实现“按需”编译。

2）一个分别编译的 Makefile 文件

上面的 Makefile 中虽然只有一行简单的编译命令，但是每次都需要编译整个工程，编译工作量很大，只适用于极简单的情况。下面给出了一个更合理一点的编译方式，可以减少很多不必要的重复编译。

```
##Makefile2##
hello: hello.o main.o
    g++ main.o hello.o -o hello

main.o：main.cpp ./include/hello.h
    g++ -I./include/ -c main.cpp -o main.o

hello.o:./src/hello.c ./include/hello.h
    gcc -I./include/ -c ./src/hello.c -o hello.o

.PHONY clean
clean:
    @echo “clean...”
    rm -rf main.o hello.o hello
```

用上面的例子进一步解释 make 执行的过程：假设前面执行过一次 make 命令，即当前工程中存在上次 make 生成的 hello hello.o main.o 三个文件。修改 hello.c 的代码，保存后执行 make（make hello）命令时，Makefile 中的流程如下：

（1）make 程序检查目标 hello 和它依赖 hello.o、main.o 的时间戳看哪个更新。由于 hello.o 和 main.o 本身也是目标，make 转去判断 hello.o 及 main.o 的依赖；

（2）hello.o 本身也是一个目标，make 程序会比较 hello.o 和其两个依赖./src/hello.c hello.h 的时间戳。由于我们更改过 hello.c 的代码，因此，hello.c 的时间戳要比 hello.o 晚。make 程序会调用下面的命令产生新的 hello.o；

```
gcc -I./include/ -c ./src/hello.c -o hello.o
```

（3）检查 main.o 和它的依赖的时间戳。由于 main.cpp 和 hello.h 都没有改动，因此 main.o 不需要重新生成；

（4）由于 hello.o 被更新了，需要返回第一步更新 hello 目标。

我们可以发现：make 程序根据 Makefile 文件中的依赖关系及其对应的编译选项，按照需要对源代码进行编译，大大减少了完全编译所需要的工作量。

进一步解释伪目标：

上面简单地说 clean 是一个伪目标，但是在实际应用中并不是很可靠，需要显式地告诉 make 程序。.PHONY clean 显式地表明 clean 是一个伪目标。显式地标记为伪目标的好处是每次执行 make clean，make 程序总会以为当前目标的时间戳早于其依赖，也就是下方的命令代码总是可以得到执行。如果没有.PHONY clean 语句，在当前 Makefile 目录下面新建一个名字叫 clean 的文件或者文件夹，make 程序会拿这个文件（文件夹）的建立时间作为 clean 的时间戳，可能使得 make clean 不能正确执行。

3）一个采用自动化变量的 Makefile

本 Makefile 中给出了静态库（archive）产生及使用的示例。

```
##Makefile3##
DefaultTarget=hello_lib

Default：$(DefaultTarget)

hello: ./src/hello.o main.o
    g++ $^ -o $@

%.o：%.cpp ./include/hello.h
    g++ -I./include/ -c $< -o $@

%.o:%.c ./include/hello.h
    gcc -I./include/ -c $< -o $@

libhello.a：./src/hello.o
    ar -r $@ $^

$(DefaultTarget):main.o libhello.a
    g++ main.o -L. -lhello -o $(DefaultTarget)
#-L.表示将当前路径添加到链接器搜索路径中；
#-lhello 表示编译的时候使用  libhello.a 库；
#GCC 编译器默认会自动为 hello 添加 lib 前缀、.a 后缀。
.PHONY clean
clean:
    rm -rf main.o ./src/hello.o hello
```

变量解释：

$^表示冒号右边的所有依赖；

$<表示冒号右边的第一个依赖；

$@表示冒号左边的目标；

%表示匹配任意字符串。

如果有少量的代码可以用第二个例子中的 Makefile 的书写方式，当工程中有大量源代码程序的时候显然很难维护，这里采用自动变量的方式大大简化了 Makefile。本 Makefile 中将第一行的 hello.o 修改为./src/hello.o，及表示产生的 hello.o 文件不再存放在当前路径下，而是放在 src 子目录下。这样做是因为将%.o:%.c ./include/hello.h 中的“%”用./src/hello 替换后可以变成：./src/hello.o:./src/hello.c ./include/hello.h，从而可以用对应的规则生成 hello.o。变量定义及使用的规则基本同 shell 脚本，不再一一细说。

4）一个可以自动识别的 Makefile

```
##Makefile4##
CC=gcc
XX=g++
AR=ar
LD=$(XX)
CFLAGS=$(foreach n,$(TPATH),-I$(n))
CFLAGS+= -m32 -march=i686 -O2
MACRO+= -DNDEBUG

TARGET=hello
LIB=libhello.a
tTPATH=./    src/    include/
PREFIX=
TPATH=$(foreach n,$(tTPATH),$(PREFIX)$(n))
SOURCES+=$(foreach n,$(TPATH),$(wildcard $(n)*.c $(n)*.cpp))
HEADERS+=$(foreach n,$(TPATH),$(wildcard $(n)*.h ))
OBJECTS=$(patsubst %.c,%.o,$(patsubst %.cpp,%.o,$(SOURCES)))

$(TARGET):$(OBJECTS)
    @echo "link..."
    $(LD) $(CFLAGS) $(MACRO) $(OBJECTS) -o $(TARGET)

%.o:%.c $(SOURCES) $(HEADERS)
    @echo "compile... "
    $(CC) $(CFLAGS) $(MACRO) -c $< -o $@

%.o:%.cpp $(SOURCES) $(HEADERS)
    @echo "compile... "
    $(XX) $(CFLAGS) $(MACRO) -c $< -o $@

lib:$(LIB)
$(LIB):$(OBJECTS)
    $(AR) -r $@    $^

.PHONY: clean
clean:
    rm -rf $(TARGET) $(OBJECTS) $(LIB)
### Makefile ends here
```

最后给出一个可以自动识别的 Makefile，理论上可以编译任何书写规范的纯 C/C++ 代码程序（可以编译但不是最合理的方案，存在重复编译），可以在此基础上稍做修改得到一个更合适的 Makefile。这个 Makefile 需要手动维护 tTPATH 中所保存的源代码的目录。

附 A：预处理用于调试

从上面的代码中可知，我们采用预处理变量来避免重复包含头文件。C++程序员有时也会使用类似的技术有条件地执行用于调试的代码。这种想法是：程序所包含的调试代码仅仅在开发过程中执行。当应用程序已经开发完成，并且准备提交时，就会将调试代码关闭。可使用 NDEBUG 预处理变量来实现有条件地调试代码：

```
int main()
{
#ifndef NDEBUG
cerr<<”starting main”<<endl;
#endif
...
}
```

如果 NDEBUG 未定义，那么程序就会将信息写到 cerr 中。如果 NDEBUG 已经定义了，那么程序执行时将会跳过#ifndef NDEBUG 和#endif 之间的代码。在开发过程中，只要保持 NDEBUG 未定义就会执行其中的调试语句。开发完成后，要将程序交付给客户时，可以通过定义 NDEBUG 预处理变量，删除调试语句。大多数编译器都提供定义 NDEBUG 的命令行选项：

```
CC -DNDEBUG main.c
```

这样的命令等效于在 main.c 的开头提供#define NDEBUG 预处理命令

NDEBUG 的一个直接的应用是 assert（断言）预处理宏。Assert 宏是在 cassert 头文件中定义的，所以使用 assert 需要包含这个头文件。Assert 宏需要一个表达式作为它的条件：

```
assert（expr）
```

只要在 NDEBUG 未定义的情况下，assert 就求解条件表达式，如果结果未 false，assert 输出信息并且终止程序的执行。如果表达式有一个非零值，则 assert 不做任何操作。

预处理器还定义了其余四种在调试中非常有用的变量：

__FILE__文件名

__LINE__当前行号

__TIME__文件被编译的时间

__DATE__文件被编译的日期

可使用这些常量在错误消息中提供更多的信息。

个人建议：简单的问题尽量用 assert 预处理宏做防错处理，复杂的采用 #ifndef NDEBUG ... #endif 包含。临时性的调试代码用#ifndef NDEBUG ... #endif 包含，以便可以方便地关闭所有调试代码。

附 B：命名空间的使用

不同人写的代码库之间放在一起用难免出现定义的函数、变量等出现重名的问题，这种名字冲突问题称为命名空间污染问题。命名空间为防止名字冲突提供了一种可控的机制。命名空间定义以关键字 namespace 开始，后接命名空间的名字。下面修改上面 hello 例子的代码，简单说明 namespace 的使用：

```
/**./include/hello.h**/
/**文件位于 include 子文件夹下**/
#ifndef HELLO_H
#define HELLO_H
namespace VPU{
   void helloFunc(char * name);
   extern int g_val;
   //内联函数的定义及声明都在头文件中
   inline void setVal（int val）{
        g_val=val;
   }；
}
#endif
#endif /* HELLO_H */

/**./src/hello.cpp**/
/**文件位于 src 子文件夹下**/
#include <stdio.h>
#include "hello.h"
//“内部函数”不需要在头文件中声明
static void _helloIngenic()
{
   std::cout<<”Hello Ingenic!”<<std::endl;
}
void VPU::helloFunc(char * name)
{
   printf("Hello %s!\n",name);
   _helloIngenic();
}

int VPU::g_val=0;


/**main.cpp**/
#include <iostream>
#include "hello.h"

using namespace std;
//using VPU::helloFunc

int main(int argc, char** argv)
{
   char name[20]="VPU";
   VPU::helloFunc(name);
   //helloFunc(name);

   return 0;
}
```

注意：由于 namespace 属于 C++的特性，这里将 hello.c 改为 hello.cpp。